Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

TRABAJO DE FIN DE GRADO

CLASIFICACIÓN DE TRÁFICO DE RED USANDO HADOOP Y GPUS

Oscar Asenjo Bennai
Tutor: Iván González Martínez

JULIO 2015

CLASIFICACIÓN DE TRÁFICO DE RED USANDO HADOOP Y GPUS

Autor: Oscar Asenjo Bennai
Tutor: Iván González Martínez

High Performance Computing and Networking
Escuela Politécnica Superior
Universidad Autónoma de Madrid

JULIO 2015

Agradecimientos

A todas las personas que me han ayudado ha llegar hasta aquí.

Abstract

Abstract — Internet is an ever-growing network. Since the 60s to the present, there has been a growth in the number of services and the bandwidth provided by the ISPs, from 56 kbps to the current 100 Gbps. For ISPs it's of great importance to classify the type of traffic flowing through their networks, whether to perform forensic diagnostics or to improve the QoS offered for their clients. The amount of data generated requires an increasingly complex and expensive software and hardware infrastructure, both in terms of storage and processing of this information.

This kind of problem in which standard software and hardware solutions do not provide enough performance to meet the computing need in a reasonable time period have led to the emergence of new solutions that try to mitigate these problems. One of these new solutions is Hadoop, an open-source software developed by Apache that uses the computers in a distributed system to split the amount of data on them, based on the MapReduce programming paradigm. Another possible solution is the use of specialized hardware for massive data computing, specifically the use of GPUs. These devices have a different hardware architecture that gives better performance that CPUs in massive data computing.

For this project, it has decided to combine these two solutions, using Hadoop as a tool for storage and distribution of data to be analyzed in different machines in a distributed system, and use on each of them a GPU to accelerate data processing.

Key words — Hadoop, MapReduce, GPU, distributed system

Resumen

Resumen — Internet es una red en constante crecimiento. Desde sus orígenes en los años 60 hasta la actualidad, se ha observado un crecimiento tanto en el número de servicios ofrecidos como en los anchos de banda proporcionados por los proveedores de servicios de internet, pasando de los 56 kbps de los módems telefónicos a las actuales redes de 100 Gbps. Para los proveedores de servicios es de una gran importancia clasificar el tipo de tráfico que circula por sus redes, ya sea para poder realizar diagnósticos forenses ante un ataque o para mejorar el QoS ofrecido. La ingente cantidad de datos que se generan provocan que sea necesario una cada vez más compleja y costosa infraestructura software y hardware, tanto a nivel de almacenamiento como de procesamiento de esta informacion.

Este tipo de problemáticas en las cuales las soluciones software y hardware estándar no proporcionan un rendimiento suficiente como para satisfacer la necesidad de cómputo en periodos de tiempo "razonables" han derivado en la aparición de nuevas soluciones que tratan de mitigar estas problemáticas. Una de estas nuevas soluciones es Hadoop, un software open-source desarrollado por Apache y que se basa en aprovechar todos los equipos de un sistema distribuido para repartir la cantidad de datos a procesar, basándose en el paradigma de programación MapReduce. Otra posible solución es el uso de hardware especializado para la computación masiva de datos, concretamente el uso de GPUs. Estos equipos hardware permiten acelerar el procesamiento de datos debido a sus características frente a las CPUs.

Para este Trabajo de Fin de Grado, se ha decidido combinar estas dos soluciones, utilizar Hadoop como herramienta de almacenamiento y distribución de los datos a analizar en las distintas máquinas de un sistema distribuido, y sobre cada uno de los equipos utilizar GPUs para acelerar el procesamiento de los datos.

Palabras clave — Hadoop, MapReduce, GPU, sistema distribuido

Índice general

Índice de tablas

Índice de figuras

1

Introducción

Internet está evolucionando cada día. Ha pasado de ser una red dedicada al intercambio de información entre universidades a una red que conecta todo el mundo, con el crecimiento que esto supone tanto a nivel de usuarios y aplicaciones como de protocolos de comunicación y paquetes de datos para intercambiar toda esa información. Este volumen de datos es cada vez más grande, suponiendo un doble problema para los proveedores de servicios de internet que necesitan clasificar el tipo de tráfico de red que gestionan sus infraestructuras. Por una parte, todos estos datos deben ser almacenados hasta ser procesados, lo que supone un problema en base a la ingente cantidad de información generada, y por otra parte toda esta información debe ser procesada en un tiempo razonable, con la complejidad que supone tanto a nivel de infraestructura hardware como de desarrollo software .

Esta problemática es la que se define como ”Big Data” o ”datos masivos”. Con estos términos se hace referencia a una cantidad de datos tal que supera la capacidad del software y hardware habitual para ser capturados, gestionados y procesados en un tiempo razonable. En la actualidad, en la que se pueden llegar a transmitir hasta 100 Gbps de datos, la cantidad de datos que se generarían diariamente serían del orden de 8640 Tb,

con las dos problemáticas definidas anteriormente.

Para dar solución al procesamiento de datos masivos han ido surgiendo soluciones que pretenden acelerar este proceso basándose, entre otras posibilidades, en la paralelización del análisis de los datos. Una de estas nuevas soluciones es Hadoop, un framework basado en el paradigma de programación MapReduce que aprovecha los equipos de un sistema distribuido para repartir los datos a procesar entre ellos, paralelizando su análisis.

A pesar de ser una solución que permite distribuir la carga de proceso entre varios nodos de computación, la mejora de rendimiento puede no ser suficiente como para ofrecer tiempos de procesamiento razonables si la cantidad de datos es extremadamente grande. Es por esto que se ha optado por tratar de combinar la ventaja que supone Hadoop con otra solución para la computación masiva: el uso de hardware especializado, concretamente el uso de GPUs. Las características hardware de una GPU permiten una mayor velocidad de procesamiento de datos que una CPU[1].

Por tanto, en este TFG se va a comprobar la viabilidad de combinar ambas soluciones, Hadoop y GPUs, para clasificar paquetes de red según su protocolo mediante la técnica de inspección profunda de paquetes.

1.1. Motivación del proyecto

La motivación de desarrollo de este proyecto es la evidencia que conforme aumenta la expansión de la red aumenta el número de datos generados y por tanto susceptibles de clasificar. En este proyecto se va a desarrollar un clasificador de paquetes de red basado en la inspección profunda de paquetes, pero no es el único campo al que se le puede aplicar una aplicación que combine ambas soluciones. Un ejemplo es el campo de las redes sociales, comercio online o publicidad personalizada, en el que se analizan ingentes cantidades de información relacionadas con usuarios para recomendar productos que le podrían interesar o recomendar amigos que podrían conocer, o el campo de las simulaciones, en el que se necesitan grandes muestras de datos para poder analizarlas y obtener modelos matemáticos concluyentes.

En este proyecto se está comprobando únicamente la viabilidad de combinar ambas tecnologías y no su rendimiento frente a otras alternativas, este análisis se deberá realizar en el futuro con distintas aplicaciones para valorar si, siendo viable esta combinación, la mejora de rendimiento que se obtiene es significativa frente a otras posibles soluciones

respecto al coste que supone tanto en infraestructura como en desarrollo software.

1.2. Objetivos

El objetivo de este proyecto es el de demostrar la viabilidad de combinar Hadoop y procesamiento sobre GPU para clasificar paquetes de red según su protocolo mediante la técnica de inspección profunda de paquetes, de forma que al final de la ejecución obtendremos una lista de protocolos junto con el número de paquetes de red que pertenecen a cada uno de ellos.

1.3. Estructura del documento

Este documento se va a estructurar con las secciones en el orden que se encuentran aquí descritas

* Introducción: Una introducción a la motivación del proyecto, los objetivos del mismo, y la estructura del documento.

* Estado del arte: Las implementaciones actuales de frameworks, aplicaciones y servicios basadas en MapReduce y GPUs.

* Análisis: Breve descripción de las herramientas utilizadas, requisitos de la aplicación y justificación de la arquitectura y el lenguaje elegidos para la implementación.

* Desarrollo: Descripción de todas las etapas de desarrollo del proyecto.

* Resultados: Validación de la implementación desarrollada y pruebas de rendimiento frente al clasificador de paquetes original.

* Conclusiones: Exposición del proyecto desarrollado, junto con las posibles posibilidades de desarrollo futuro.

2

Estado del Arte

2.1. Introducción

El framework Hadoop se encuentra basado en el modelo de programación MapReduce, desarrollado por Google en 2004[2]. El hecho de combinar soluciones basadas en MapReduce sobre GPUs no es algo nuevo. Es visible el potencial que tiene el combinar un framework que distribuya de forma transparente la carga de proceso entre distintos nodos de computación, liberando al desarrollador del coste que supone la gestión de recursos, con la aceleración del procesamiento de esos datos sobre una GPU, cuyas características físicas permiten llegar a obtener unas extremadamente significativas mejoras de rendimiento [3] comparado con aplicaciones ejecutadas sobre CPU. Es por esto por lo que muchas universidades han desarrollado aplicaciones y estudios propios sobre la combinación de aplicaciones basadas en MapReduce con GPUs, ya sea sobre CUDA, OpenCL u otros lenguajes [4][5] . Aparte de estos desarrollos para proyecto concretos, existen diferentes plataformas y servicios de pago que combinan frameworks distribuidos junto con GPUs para acelerar el procesamiento.

2.2. MapReduce y GPUs

En esta sección se van a describir frameworks y servicios que combinan aplicaciones basadas en MapReduce con GPUs.

2.2.1. Mars

El framework Mars[6] se desarrolló en el año 2008, y se trata de un framework codificado en C/C++ y CUDA que ejecuta el algoritmo MapReduce directamente sobre la GPU. Este framework únicamente es compatible con gráficas de NVIDIA ya que se encuentra desarrollado únicamente en lenguaje CUDA.

La mejora de rendimiento que proporciona este framework sobre el framework Phoenix[7], que implementa el algoritmo MapReduce sobre procesadores multicores, es de entre el x1.5 y el x16 en función de la aplicación ejecutada.

2.2.2. MapCG

MapCG[8] es un framework desarrollado en 2010 y combina las GPUs y los procesadores multicores mediante la filosofía MapReduce. Este framework, cuyas primitivas están escritas en CUDA y por tanto delimitan su uso a equipos con GPUs de NVIDIA, permite ejecutar el algoritmo MapReduce sobre GPU o procesadores multicore, éstos últimos a través de la librería de programación multiproceso OpenMP[9], y también permite ejecutar el algoritmo simultáneamente en ambas plataformas.

Este framework consigue en su versión multicore una aceleración frente a Phoenix-2[10], la revisión del framework Phoenix ejecutado únicamente sobre CPUs multicore, de entre el x2-3. En su versión GPU frente al framework Mars obtiene una aceleración de x1.6-2.4 en función de la cantidad de datos a procesar.

2.2.3. Amazon Elastic MapReduce

Se trata de un servicio web desarrollado y gestionado por Amazon que se basa en Hadoop y que se ejecuta en un clúster de servidores virtuales en Amazon Elastic Compute

Cloud[11], su propia nube de servidores. Desde 2010, los servidores usados por Amazon soportan programación sobre GPUs[12].

2.3. Java y GPUs

Hadoop permite desarrollar las aplicaciones según la filosofía de MapReduce en diferentes lenguajes diferentes[13]. Para el lenguaje de programación Java, sobre el que se va a desarrollar la aplicación Hadoop, existen distintas alternativas a la hora de ejecutar funcionalidades sobre GPU

2.3.1. JCuda

JCUDA[14] es una librería desarrollada en Java que permite ejecutar aplicaciones sobre GPUs de NVIDIA. Posee dos funcionalidades principalmente: por una parte, se usa como interfaz para ejecutar las librerías de tiempo de ejecución de CUDA, como son JCublas o JCuff. Por otra parte, permite ejecutar kernels que previamente han sido programados y compilados en CUDA con su compilador propio.

2.3.2. Java Aparapi

Se trata de una librería desarrollada en 2011 por AMD y posteriormente convertida en un proyecto Open Source, y que se encuentra respaldada por un gran número de fabricantes, entre los que se incluyen Intel, Xilinx, NVIDIA o Altera[15]. Está librería convierte en tiempo de ejecución el código compilado Java en código OpenCL y lo ejecuta sobre GPU. En caso de que no exista ninguna GPU compatible instalada o falle la ejecución en ella, el código se ejecuta a través de threads.

A diferencia de JCUDA, no necesita una compilación previa de los kernels de la GPU.

2.3.3. JNI

JNI son la siglas de Java Native Interface, un framework que permite la ejecución de funciones nativas desde la JVM (Java Virtual Machine)[16]. A diferencia de las librerías descritas anteriormente, JNI no es un framework dedicado en exclusiva a la computación

paralela, si no que se trata de una interfaz que soluciona la principal carencia de Java: su lenguaje a alto nivel, que le impide ejecutar funciones del sistema a bajo nivel.

JNI proporciona una interfaz a través de la cual se pueden ejecutar desde Java funciones de C/C++ y ensamblador, que pueden ejecutarse sobre CPU o sobre GPU si el código nativo contiene funcionalidades codificadas en CUDA u OpenGL. Por tanto, aunque su finalidad no es exclusivamente la computación paralela, proporciona una ventana para éste propósito.

2.4. Conclusión

Se puede comprobar que la combinación de algoritmos MapReduce con GPUs, ya sea para acelerar el procesamiento de los datos en alguna de las fases o para ejecutar enteramente el algoritmo sobre la GPU es una opción que ha suscitado mucho interés por sus posibles aplicaciones en necesidades de computación masiva. Además, se ha comprobado que los frameworks que aprovechan el potencial de procesamiento de la GPU obtienen mejor rendimiento que frameworks de características similares ejecutados únicamente sobre CPUs multicore.

3

Diseño

3.1. Tecnologías utilizadas

Este proyecto ha verificado la viabilidad de combinar el framework Apache Hadoop, basado en el modelo de programación MapReduce, junto con el procesamiento sobre GPUs para acelerar el análisis de los datos.

3.1.1. MapReduce y Apache Hadoop

MapReduce es un modelo de programación orientado al procesamiento de grandes cantidades de datos en entornos distribuidos, tanto en clústers de computadores como en GPUs o procesadores multicore, originalmente desarrollado por Google[2]. Este modelo de programación se basa en el uso de las funciones Map y Reduce para abordar el análisis de los datos y en el uso de pares clave-valor para distribuir los datos entre la fase Map y Reduce, lo que debe tenerse en cuenta a la hora de optar por MapReduce como solución al desarrollo de la aplicación ya que no todas las aplicaciones pueden ajustarse a este

modelo.

En la arquitectura del algoritmo MapReduce, los datos de entrada son ”trozeados” (splits) y distribuidos por los distintos nodos de computación. Cada uno de estos splits se convierte en un par clave-valor en base al algoritmo de asignación de claves utilizado por el lector de datos de entrada, y se ejecuta en una función Map, que se encarga de procesar los datos y reasignar pares clave-valor en función de las necesidades de la aplicación. Entre el proceso Map y el proceso Reduce, los valores se "mezclan"(shuffle), ordenándose en función de la clave que se les ha asignado en la fase Map. Una vez realizada esta ordenación, los datos se pasan a la función Reduce, que los procesa en función de la clave asignada en la función Map y es la encargada de producir la salida de la aplicación.

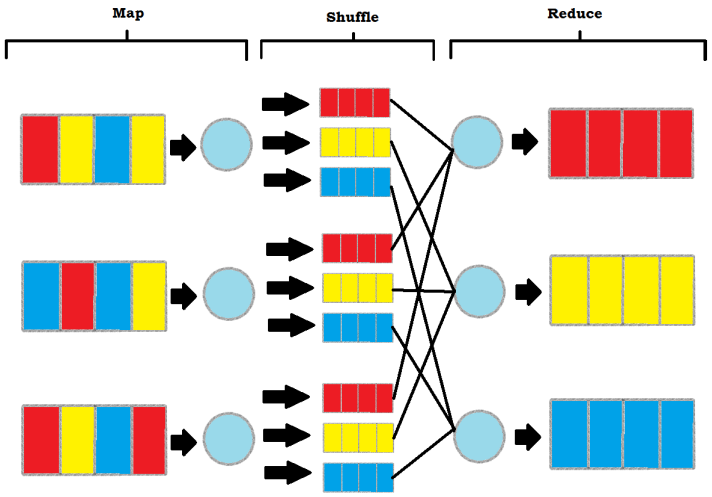

Figura 3.1: Arquitectura de MapReduce

Hadoop es un framework basado en MapReduce, desarrollado inicialmente por Yahoo! y mantenido en la actualidad por la comunidad Apache a través de una licencia de software libre[17]. Utiliza un sistema de archivos distribuido denominado HDFS (Hadoop Distributed File System), que ofrece fiabilidad en los datos almacenados debido a su replicación en múltiples nodos pero, sin embargo, no proporciona Alta disponibilidad de los datos. Es en este sistema de archivos distribuidos donde se copian tanto los datos de entrada de una aplicación Hadoop como la salida resultante si finaliza con éxito el trabajo.

El funcionamiento de Hadoop se basa en el Job Tracker y el Task Tracker. El Job Tracker es el servicio sobre el que se lanza un trabajo Hadoop y que asigna un determinado trabajo a los TaskTracker, que son los nodos del clúster Hadoop que están habilitados en

ese momento para realizar tareas. El criterio para decidir que TaskTracker es asignado a la aplicación lanzada es la cercanía a los datos que van a procesarse, intentando que se encuentren en la misma máquina o en el mismo rack, de forma que la transmisión de los datos desde el equipo en el que se encuentran hasta el que los requiere para procesarlos no penalice el tiempo de ejecución del trabajo y tratando de evitar errores derivados de la transmisión de los datos. El JobTracker es el que se encarga de coordinar los diferentes TaskTracker y supervisar que finalizan correctamente su tarea asignada, abortando la aplicación en caso de ser necesario. Este framework ofrece cierta tolerancia a errores, de forma que si un nodo falla en la ejecución del trabajo, la tarea que tenía asignada es derivada a otro Task Tracker que en ese momento esté disponible para ejecutar trabajos.

Cada uno de los TaskTracker ejecuta una fase del algoritmo MapReduce, permitiendo ejecutar en paralelo en distintas máquinas las funciones Map y Reduce.

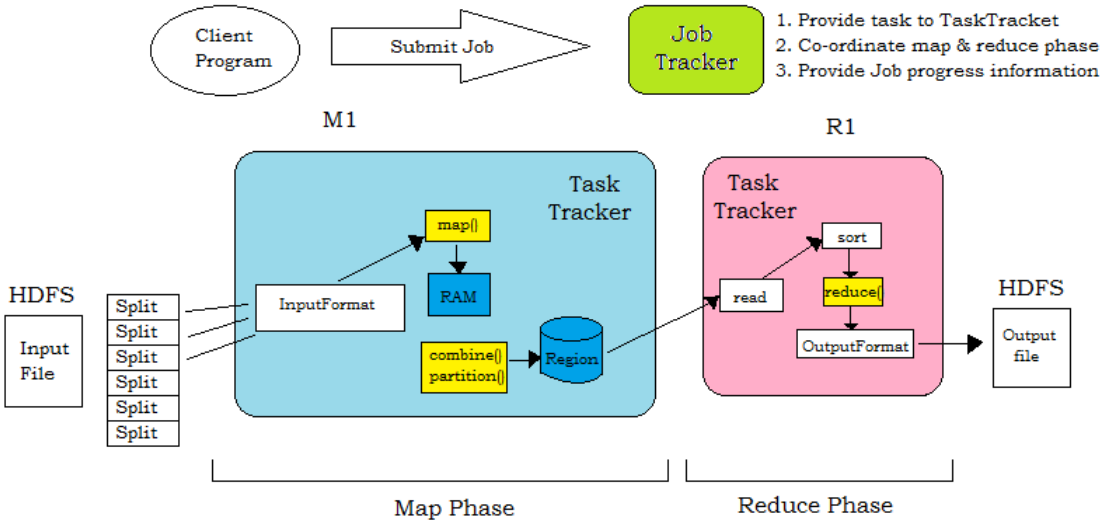

Figura 3.2: Arquitectura de Hadoop

3.1.2. GPU y CUDA

Una GPU o Unidad de procesamiento gráfico (Graphics Processing Unit) es un coprocesador presente en las tarjetas gráficas dedicado al procesamiento de gráficos y de operaciones de carga flotante, aligerando la carga de operaciones aritméticas en la CPU y permitiéndola dedicarse a otras necesidades de computación.

La principal diferencia con respecto a una CPU es que la GPU es un dispositivo

específico diseñado para el cálculo de valores en coma flotante mientras que la CPU es un dispositivo de propósito general, lo que la da un mejor rendimiento a la hora de realizar este tipo de operaciones. A su vez, debido a que la finalidad principal de una GPU es la de ejecución de operaciones en coma flotante y estas operaciones pueden ser independientes unas de otras, las GPUs están diseñadas con un gran número de núcleos que permiten la paralelización del procesamiento de los datos, acelerando el procesamiento. Conviene diferenciar entre un núcleo de CPU, que equivale a un microprocesador completo, con un núcleo GPU, compuesto básicamente por unidades aritmético-lógicas[18].

Estas dos características, la especialización del dispositivo y su alta capacidad paralela, ha devenido en lo que se conoce como GPGPU (General Purpose Computing on Graphics Processing Units), es decir, aprovechar las capacidades de alto rendimiento de las GPU para la programación de propósito general, más concretamente, el procesamiento en la GPU de algoritmos paralelizables para mejorar el rendimiento general de la aplicación [1]. En este proyecto se va a utilizar CUDA (Compute Unified Device Architecture), un lenguaje con su compilador y conjunto de herramientas asociadas que permiten desarrollar aplicaciones que se ejecuten sobre GPUs de NVIDIA.

En la ejecución de un código sobre la GPU a través de la API de NVIDIA, inicialmente es necesario reservar memoria y copiar los datos que se van a procesar. Después, desde el código principal se ejecuta la llamada a la GPU, la cual realiza el procesamiento de la información y el resultado lo escribe en su memoria para que sea leído desde la CPU. Esta transmisión de información entre la GPU y la CPU añade cierto coste extra de tiempo de computación (overhead), lo que implica que el uso de GPUs para procesamiento de datos no es recomendable en cualquier circunstancia, únicamente cuando el ahorro obtenido por la mayor velocidad de procesamiento supere a los costes de transmisión de datos entre la CPU y la GPU[19].

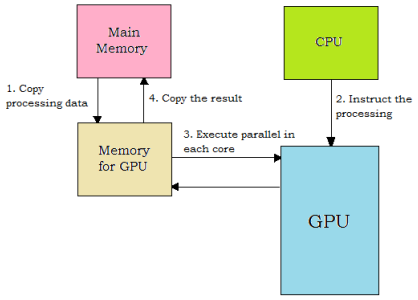

Figura 3.3: GPGPU

3.1.3. Motor DPI

El motor DPI (DPIEngine), al que también nos referiremos en este documento como "clasificador de paquetes", es una aplicación codificada en C/C++ y CUDA y es la encargada del análisis y clasificación de los paquetes de red mediante la técnica DPI (Deep Packet Inspection), en la cual se utiliza la carga del paquete a analizar y no la cabecera.

A su vez, incluye la aplicación FlowProcess, basada en la aplicación NetFlow desarrollada por Cisco y que permite generar flujos de red en base a un fichero de trazas de red obtenido mediante un analizador.

Para realizar la clasificación se comprueba si un conjunto de bytes de un flujo de paquetes de red, generado con la aplicación FlowProcess y correspondientes a la carga útil de un paquete, cumplen una cierta expresión regular asociada a un protocolo. Las expresiones regulares utilizadas en esta aplicación se han obtenido de l7-filter[20]. El clasificador de paquetes genera para cada ocho expresiones regulares asociadas a un protocolo de red una DFA (autómata finito determinista), y posteriormente en la GPU va realizando sobre los paquetes de red el análisis de los protocolos en función de las DFA cargadas. La salida de este programa está desglosada por paquetes, donde se muestra su protocolo en el caso de que haya existido una correspondencia con alguna de las expresiones regulares.

Ambas aplicaciones se encuentran definidas y descritas en [21]. La funcionalidad ha sido tratada como una caja negra, únicamente se han hecho modificaciones y adaptaciones en la forma de gestionar las entradas y salidas de datos de la aplicación para funcionar a través de la interfaz JNI.

3.2. Análisis y planificación

3.2.1. Requisitos funcionales

El objetivo de este proyecto es comprobar la viabilidad de combinar Hadoop y GPUs para desarrollar un clasificador de paquetes de red, por tanto el único requisito funcional es que la aplicación se ejecute correctamente. La forma de validar la correcta ejecución de la aplicación es que la salida de la aplicación Hadoop debe ser similar a la del motor DPI

para un mismo flujo de red procesado.

3.2.2. Requisitos no funcionales

La gran ventaja del uso de Hadoop es que este framework hace transparente a ojos del desarrollador toda la gestión de recursos del clúster, asignación de memoria, comunicación entre nodos y seguridad en las comunicaciones, de forma que la única tarea del desarrollador es programar la configuración del trabajo y las funciones Map y Reduce[17]. Por tanto el único requisito no funcional que se ha considerado en el desarrollo de esta aplicación es el uso de la última versión liberada de la API de Hadoop, de forma que la aplicación implemente las últimas actualizaciones y funcionalidades del framework. En el comienzo del desarrollo de este proyecto, la última versión de la API era la 2.5.0.

3.2.3. Metodología de desarrollo

A la hora de planificar este proyecto se ha optado por usar un desarrollo iterativo e incremental[22].

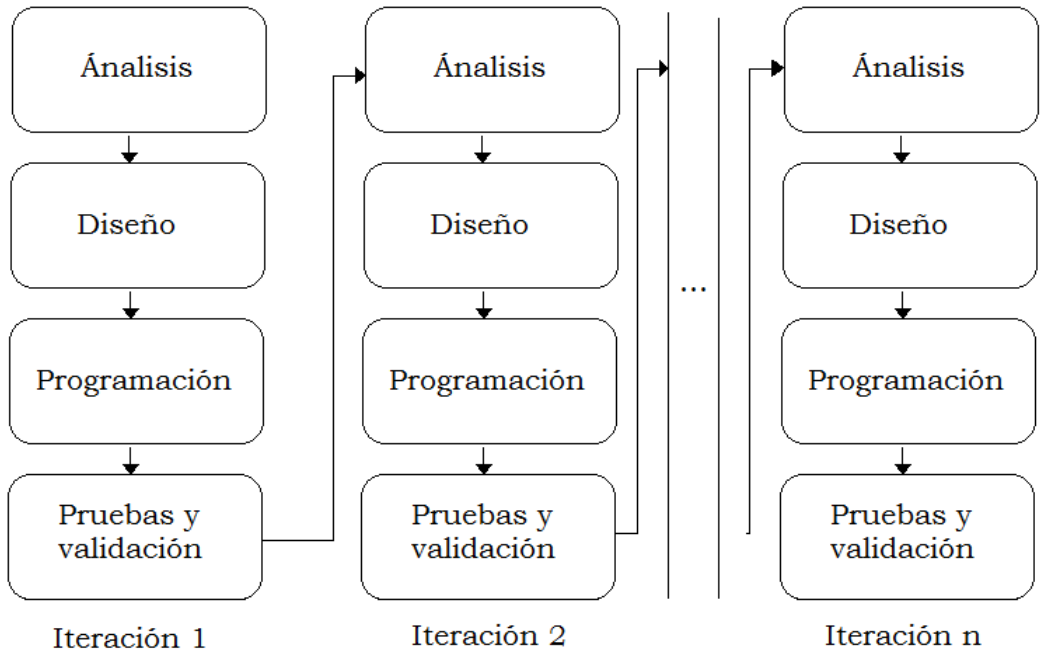

Figura 3.4: Modelo incremental iterativo

Este modelo consiste en, utilizando como base una aplicación con una funcionalidad básica, ir realizando incrementos de forma iterativa en las que en cada uno de ellos se vayan introduciendo nuevas funcionalidades; este proceso se repite hasta que se ha obtenido la funcionalidad final de la aplicación.

Debido a sus carácterísticas, se ha optado por dividir la realización del proyecto en tres desarrollos:

- El primer desarrollo ha estado orientado a realizar una aplicación basada en Hadoop y que sea capaz de procesar archivos binarios a través de la interfaz JNI.

- El segundo desarrollo consiste en diseñar una aplicación en Java que a través de la interfaz JNI ejecute el motor DPI para analizar un flujo de red.

- El tercer desarrollo ha consistido en, una vez finalizados los dos desarrollos anteriores, realizar la integración de ambas soluciones para poder ejecutar el clasificador de flujos de red a través del framework Hadoop.

Para cada uno de los tres desarrollos se han mantenido reuniones semanales con el tutor en las cuales se ponía en común la situación del proyecto en ese momento con los avances que se habían realizado en la última semana, los problemas encontrados y como se habían resuelto en caso de haberlo conseguido o con posibles soluciones en caso de estar bloqueados, y se definían los hitos y las prioridades para la siguiente semana, en base a los cuales se definía el próximo incremento a desarrollar.

3.3. Arquitectura de la aplicación

Después de analizar el funcionamiento del framework de Hadoop y del motor DPI, se decidió que la mejor forma de combinar ambas tecnologías era ejecutar el motor DPI en la fase Mapper del trabajo Hadoop, de forma que la arquitectura de la aplicación a desarrollar se configuró de la siguiente forma:

Tomando como base un fichero de tipo pcap obtenido a través de un analizador de tráfico de red (para este proyecto se utilizó Wireshark), se parsea con la aplicación FlowProcess[21] para generar flujos de red. Estos flujos se cargarán en el HDFS para poder clasificarlos a través del trabajo Hadoop. El trabajo Hadoop permite configurar el tamaño de los splits que se le pasan a los Mapper, pero puesto que es en esa fase donde se va a ejecutar el procesamiento sobre GPU se debe de tratar de que el tamaño del split

sea suficientemente grande como para que la pérdida de rendimiento por la comunicación entre GPU y CPU no sea mayor que la aceleración de procesamiento de la GPU. La función Map de Hadoop llamará a la librería de CUDA a través de la GPU, y utilizará a la hora de transmitir los datos a la fase Reduce el nombre del protocolo como clave y el número de veces que se ha encontrado en el split recibido como valor, de forma que la fase Reduce únicamente tiene que sumar los valores asociados a cada clave para obtener el número de veces que se ha encontrado un protocolo en los paquetes de red.

En cuanto al lenguaje utilizado para desarrollar la aplicación en Hadoop, el framework permite ejecutar aplicaciones en diferentes lenguajes de programación, de los que los más utilizados son:

- Java

- C++, a través del modo Hadoop Pipes[23].

- Python, a través del modo Hadoop Streaming[24]. Hadoop Streaming también permite ejecutar trabajos Hadoop en otros lenguajes.

Después de una valoración, se decidió desarrollar la aplicación en el lenguaje Java por dos motivos principales:

- La API original de Hadoop se encuentra desarrollada en Java, y tanto la documentación como el soporte que se encuentra es más extenso en este lenguaje de programación.

- Pruebas de rendimiento realizadas sobre los tres lenguajes dan una mayor puntuación a un aplicación desarrollada en Java sobre la misma aplicación desarrollada en Python o C++, siendo el doble de rápido que la misma aplicación desarrollada en C++ y hasta nueve veces más rápida que la misma aplicación desarrollada en Python[25].

4

Desarrollo

El desarrollo del proyecto se ha realizado en tres etapas: la primera ha consistido en el desarrollo de una aplicación Hadoop que fuese capaz de recibir como datos de entrada ficheros binarios y procesarlos a través de la interfaz JNI, la segunda ha consistido en el desarrollo de una funcionalidad Java basada en JNI y que fuese capaz de utilizar el motor DPI en la máquina local, y la tercera etapa ha consistido en la integración de ambas soluciones en una única funcionalidad completa.

4.1. Entorno de desarrollo

En este proyecto se han utilizado dos entornos de desarrollo:

-Para el desarrollo de funcionalidades Hadoop que no necesitaban hacer uso de GPU se ha utilizado una máquina virtual de Cloudera sobre VMWare. Esta imagen proporciona un entorno basado en la distribución Linux Fedora con un nodo Hadoop, un gestor de nodos Cloudera, la API de Hadoop y el IDE (entorno de desarrollo integrado) Eclipse. La versión de CDH es 5.3.2 y la versión de la API de Hadoop 2.5.0.

-Todas las funcionalidades ya sean de Hadoop o de Java con JNI que hiciesen uso de GPU se han desarrollado sobre el clúster de la Escuela Politécnica Superior, compuesto por tres nodos de computación cada uno con las siguientes características:

2 GPUs por nodo: GeForce GTX 480

Intel(R) Core(TM)2 Quad CPU Q9450 @ 2.66GHz

CentOS release 6.3

Hadoop 2.5.0

CDH 5.3.2

4.2. Desarrollo en Hadoop

4.2.1. Wordcount

Se utilizó la aplicación Wordcount, una aplicación que cuenta el número de veces que se repite cada palabra dentro de un texto, como punto de partida para trabajar sobre Hadoop por varios motivos:

- El Wordcount en Hadoop es el equivalente a un "HelloWorld!" en muchos lenguajes de programación, por lo que obtener el código fuente no supone ningún problema y es un buen punto de partida para aprender como se configura una aplicación Hadoop.

- La aplicación definitiva a nivel de arquitectura va a ser muy parecida al Wordcount, ya que los datos que se reciban en fase Map se pasarán a la GPU y se retornarán los protocolos encontrados para que sean contados en la fase Reduce, por lo que la arquitectura de la aplicación no difiere mucho del Wordcount original.

El procedimiento para ejecutar una aplicación Hadoop es el siguiente:

- Se compila la aplicación con la API de Hadoop, generando un fichero .jar que será el que se ejecute por el framework.

- Se crean dos directorios dentro del HDFS (Hadoop Distributed File System), uno en el que se cargarán los ficheros que el trabajo Hadoop leerá como datos de entrada, y otro donde se volcarán los resultados en caso de finalizar correctamente el trabajo.

- Se ejecuta el .jar, pasando los argumentos que sean necesarios. En esta implementación, es necesario pasar como argumentos los directorios de entrada y salida de datos dentro del HDFS.

- Se muestra en la terminal todo el progreso del trabajo y si finaliza correctamente se encontrarán en el directorio de salida los ficheros resultantes del procesamiento, junto con un resumen de la información del trabajo que se ha ejecutado. Si se desea ejecutar nuevamente el trabajo se deberán borrar todos los ficheros del directorio de salida.

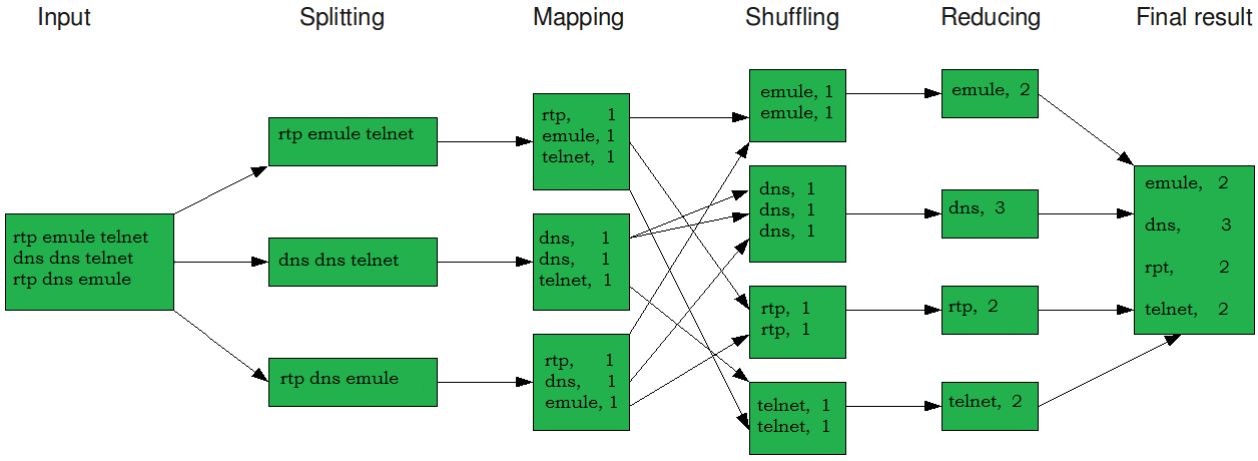

Figura 4.1: Diagrama de ejecución de WordCount

4.2.2. Lectura de datos binarios

Hadoop gestiona la entrada de datos a las funciones Map a través de ”splits”(trozos) de los ficheros de entrada. El tamaño de estos ”splits” se encuentra definido en la configuración del trabajo, y son los que se asignan a la función Map para procesarlos. Sin embargo, la función Map no procesa el ”split” entero, si no que en función del tipo de clase que esté gestionando la entrada de datos del trabajo se generan ”Records” (subcadenas) de ese ”split”. Son estos ”Records” los que se procesan en la función Map.

El framework Hadoop posee interfaces específicas para gestionar los formatos de los datos de entrada y salida del trabajo. La interfaz que deben implementar todas las clases que se encargen de gestionar el tipo de datos de entrada del trabajo es la interfaz ”InputFormat”. En la API de Hadoop se define la clase abstracta ”FileInputFormat”, que ya implementa la interfaz, y se definen también varias clases que heredan de ”FileInputFormat” y que ofrecen distintas formas de gestionar los datos de entrada del

trabajo:

- TextInputFormat: El fichero de entrada, que debe encontrarse en texto plano, se divide por líneas, siendo delimitada cada una por un salto de línea o un salto de carro. Cada línea es un ”Record” del ”split” de datos. La clave es el offset de la línea en el fichero y el valor es la propia línea de texto.

- NLineInputFormat: Es similar a la clase TextInputFormat excepto por que esta clase permite al desarrollador configurar el número de líneas que determinan un ”Record”, a diferencia de la clase TextInputFormat en la que cada ”Record” está compuesto solo por una línea. La asignación clave-valor es similar a TextInputFormat.

- SequenceFileInputFormat: El fichero de entrada es del tipo SequenceFile, un tipo de fichero específico de Hadoop.

- CombineFileInputFormat: Clase abstracta en la cuál los ”splits” que se realizan sobre los datos de entrada son de tipo CombineFileSplit, es decir, un ”split” puede tener datos de distintos ficheros de entrada.

- KeyValueTextInputFormat: El fichero de entrada se lee y se divide de forma similar a la clase TextInputFormat, sin embargo en cada línea debe existir un carácter delimitador que delimite la clave y el valor de la línea que se pasa a la fase Map, a diferencia de la clase TextInputFormat en la cuál la clave es la posición de la línea en el fichero y el valor es la línea en sí.

- FixedLengthInputFormat: Esta clase permite definir el tamaño de los ”Records” que se van a leer del ”split” de datos del Map. Como ventaja, permite el procesamiento de ficheros binarios, sin embargo tiene la limitación de que los datos a procesar deben ser de un tamaño fijo.

- Se permite al programador desarrollar su propia clase para gestionar los datos de entrada, siempre y cuando la clase desarrollada implemente la interfaz ”InputFormat” o herede de la clase abstracta ”FileInputFormat” o de cualquiera de sus subclases también abstractas. Esta solución requiere unos conocimientos del framework y de la arquitectura Hadoop muy avanzados.

Los flujos que van a clasificarse se encuentran en formato binario, lo que implica que debe usarse la clase FixedLengthInputFormat o desarrollar una clase propia para gestionar los datos de entrada. Esta última opción requiere unos conocimientos muy amplios del framework y la arquitectura de Hadoop, suponiendo un esfuerzo muy importante e

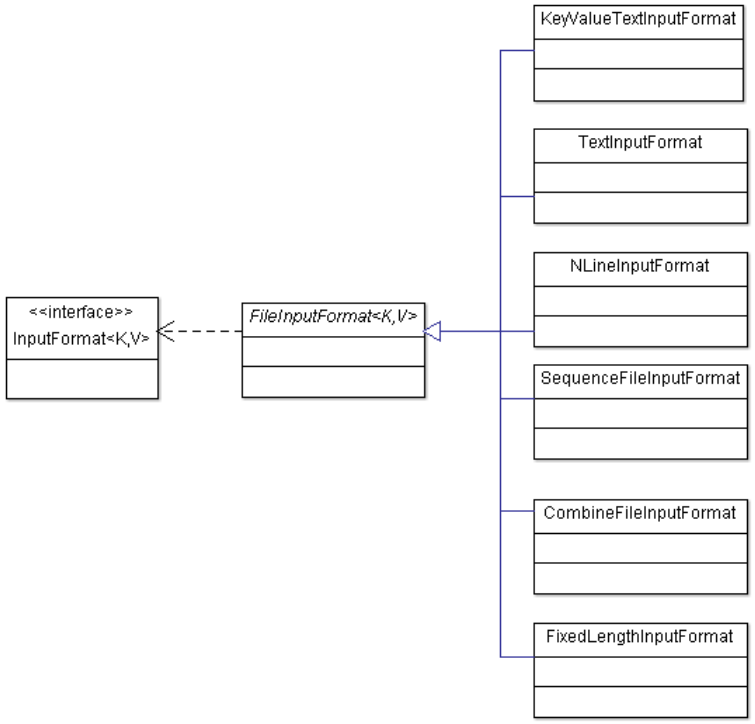

Figura 4.2: Entradas de datos de Hadoop

innecesario para el desarrollador existiendo ya una clase propia de la API que se encarga de procesar este tipo de dato de entrada, por lo que se utiliza la clase ya existente. Una limitación al realizar la adaptación de la aplicación para usar este tipo de dato de entrada es que en la versión de la API de Hadoop instalada no se encontraba disponible esta clase, lo que conllevó la descarga del código fuente y todas las dependencias de la clase para añadirla como si fuese un componente externo de la API de Hadoop.

Junto a la modificación del tipo de dato de entrada, es necesario modificar la función Map. El prototipo de la función Map definido por la API de Hadoop es "Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>", en la cuál los tipos de datos de cada uno de ellos será:

- KEYIN: La clave que asigna la función FixedLengthInputFormat al hacer el ”Record” de los datos de entrada está relacionada con la posición en el fichero del ”Record”, que es de tipo Long, por lo que el tipo de datos será LongWritable.

- VALUEIN: Puesto que los datos de entrada son binarios, el valor del ”Record” de entrada al Map será de tipo BytesWritable.

- KEYOUT: El valor de la clave que se pasará al reduce será el nombre del protocolo

encontrado en los flujos de red, por lo que será de tipo Text.

- VALUEOUT: El valor asociado a un protocolo de red será las veces que se ha encontrado en el ”split” que acaba de procesar el Mapper, por lo que deberá ser de tipo LongWritable.

Una limitación detectada en la clase FixedLengthInputFormat es que un trabajo Hadoop no puede ejecutarse si el tamaño definido para los datos no es múltiplo del tamaño total del fichero. Se intentó corregir esta limitación en el código fuente pero su complejidad lo hizo inviable, por lo que esta limitación se tuvo en cuenta para el resto del desarrollo.

En cuanto a los datos de salida del trabajo, como lo que se va a generar es un fichero de texto, se utiliza la clase de Hadoop ”TextOutputFormat” que hereda de la clase ”FileOutputFormat” e implementa la interfaz ”OutputFormat”.

4.2.3. Hadoop y JNI

La interfaz JNI (Java Native Interface) es un framework que permite a la JVM (Java Virtual Machine) ejecutar funcionalidades de otros lenguajes para solventar las limitaciones del propio lenguaje Java[16]. Para este proyecto, en el que el clasificador de flujos se encuentra codificado en C/C++/CUDA, la metodología para implementar funcionalidades a través de la interfaz nativa es la siguiente:

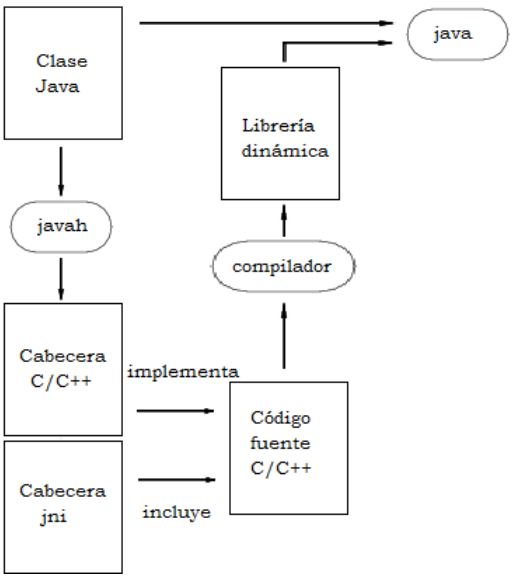

Figura 4.3: Diagrama de implementación a través de JNI

- Se debe declarar en la clase de Java la función que va a ejecutarse a través de JNI, que deberá llevar el identificador 'native' en el prototipo de la función.

- Una vez compilada la clase y generado el fichero .class, se obtiene el fichero de cabecera donde se describe el prototipo de la función que será ejecutada por la JVM.

- Se genera una librería dinámica que debe contener, como mínimo, una función cuyo propotipo coincida con el que ejecutará la JVM a través de la interfaz nativa.

- En el código fuente Java se debe cargar la librería dinámica para que pueda ser ejecutada la función nativa.

Para abstraer toda la funcionalidad Java que sirve como interfaz con JNI de la funcionalidad de Hadoop, se desarrolló una clase cuya única función es la de servir de Wrapper entre Hadoop y C definiendo el prototipo de la función nativa y cargando la librería dinámica en la que está definida la función. Esta clase se instancia en el Map del trabajo para tener acceso a la funcionalidad nativa.

Sin embargo, a la hora de utilizar una librería dinámica ajena al sistema operativo en Hadoop surge un problema: al tratarse Hadoop de un entorno distribuido en el que se desconoce a priori los nodos que van a ejecutar el trabajo, debemos asegurarnos de que la librería dinámica va a encontrarse en todos ellos. Para ello, después de intentar implementar soluciones de la API antigua de Hadoop y que ya se encuentran deprecadas, se optó por modificar el configurador del trabajo para implementar la herramienta ToolRunner de Hadoop.

Esta herramienta trabaja con el denominado 'GenericOptionsParser' y permite que al lanzar un trabajo de Hadoop se le puedan pasar argumentos para modificar ciertos parametros de la configuración del clúster, como los parámetros con los que se generan las JVM, o cargar ficheros en el HDFS. Esta última opción es la que permite copiar a todos los nodos que ejecutan el trabajo los ficheros que se le pasen por argumento despues del identificador '-files'. De esta forma, se solucionan todos los errores provocados por intentar cargar la librería dinámica en el trabajo cuando no se encuentra copiada en el HDFS y los nodos no tienen acceso a ella.

4.2.4. Validación de la adaptación

En esta fase del proyecto no se buscaba una funcionalidad de análisis de flujos si no que únicamente se buscaba validar la combinación de Hadoop y JNI. Para ello:

- Se declaró una función nativa en la clase wrapper que recibe por parámetro un array de datos binarios y devuelve un número entero.

- Se desarrolló una funcionalidad en C y JNI en la cuál únicamente se comprobaba que los datos introducidos iban de 1 a 250 en binario, retornando uno o cero en función de si los datos coincidían o no.

- Se desarrolló un programa en C que generaba un fichero con datos del 1 al 250 en binario, que se usaría como fichero de entrada.

La validación consistió en, usando un tamaño de "Record" similar al tamaño de los datos de entrada para asegurar que todos los datos leídos se procesan a la vez en la función Map, comprobar que los datos recibidos en la función a través de JNI coinciden con los datos de entrada del Hadoop. Despúes de ejecutar esta validación, se comprobó que la función retornaba correctamente a través de JNI, validando la implementación.

4.3. Java y motor DPI

El clasificador de flujos inicialmente se desarrolló como una aplicación independiente[21], pero para poder utilizarse a través de la interfaz JNI es necesario adaptarla para compilarse como una librería dinámica.

4.3.1. Adaptacion de Motor DPI

El funcionalidad del motor DPI ha sido considerada para este proyecto como una caja negra, sin embargo como esta aplicación fue desarrollada como una aplicación independiente es necesario adaptar las entradas y salidas para que pueda ejecutarse como una función a través de una llamada mediante JNI. Para ello, se debe analizar como gestiona la aplicación las entradas y salidas de datos y realizar las modificaciones necesarias.

La aplicación original recibe los siguientes parámetros:

- Un fichero binario donde se encuentran los flujos que se van a analizar. Estos flujos se generan mediante ficheros de tipo pcap que han sido posteriormente tratados con la aplicación FlowProcess. El programa va leyendo flujos de red del fichero hasta que tiene datos suficientes como para analizarlos, y una vez clasificados los flujos, vuelve a leer del fichero, este proceso se repite hasta que se han clasificado todos los datos del fichero.

- Un nombre de fichero de salida que se utiliza como base para generar dos ficheros, uno de texto y otro binario. Para cada 8 protocolos que se analizan, se genera una DFA en la que se marca a nivel de bit si alguno de los protocolos ha sido encontrado en el paquete analizado, y una vez finalizado el análisis se generan los ficheros donde se muestra para cada uno de los paquetes del flujo su protocolo en caso de que haya existido correspondencia .

- Un fichero que contiene los nombres de los protocolos que se van a utilizar para clasificar los flujos de red.

- A su vez, aunque no se reciba por parámetro, en el directorio del ejecutable debe encontrarse un fichero de configuración con la configuración del clasificador de paquetes y un directorio donde se encuentran las DFA o las expresiones regulares. En el caso de que las DFA ya existan, se cargan en memoria para preparar el análisis, si no, se crean en base a las expresiones regulares.

La API de Hadoop permite cargar ficheros en el HDFS a través de la herramienta ToolRunner, lo que implica que todos los ficheros de configuración pueden cargarse desde el trabajo y no es necesario modificar en ese sentido el motor DPI. Sin embargo, la entrada y salida de datos si que debe de ser adaptada. Además, para abstraer la comunicación entre el motor DPI y la aplicación en Java, se desarrolló una función en C que haría de interfaz entre las dos funcionalidades, recibiendo el array binario desde la función en Java y procesandola mediante las funciones JNI para llamar al motor DPI, y procesando el retorno de la función para adaptarlo a las funciones de JNI y devolverlo a la aplicación de Hadoop.

En base a éstas características de la aplicación, el planteamiento para adaptarla a las necesidades de la interfaz JNI es la siguiente: la función recibirá tres parámetros, que serán los flujos de red en formato binario y su longitud y un puntero que almacenará el número de protocolos que se están clasificando, y a su vez en el directorio donde se encuentre la librería dinámica se deberá encontrar el fichero de configuración del motor DPI. Estos flujos se clasificarán, y como salida de la función tendremos un array de tipo

de dato 'long'. Se ha elegido este tipo de dato ya que su capacidad de representación nos asegura que aunque un protocolo se repita un gran número de eveces en un flujo de red de mucho tamaño, no genere overflow en la variable y se falsee el cálculo de los datos. Como hemos visto, unos de los parámetros de entrada de la aplicación original es un fichero con los nombres de los protocolos, el orden en el que aparecen estos nombres en el fichero es el que utilizará para relacionar con los valores del array retornados de la función (el valor del primer valor del array equivale al primer nombre en el fichero, etc.). El flujo de red, que en la aplicación original se iba leyendo de fichero, en esta implementación se almacenará enteramente en memoria.

De esta forma, los prototipos de las funciones serían:

Para la función nativa:

```
native long[] generaEntrada(byte[] msg);
```

Para la interfaz entre Hadoop y motor DPI:

```
JNIEXPORT jlongArray JNICALL Java_Readbinary_generaEntrada(JNIEnv *
    ↪ env, jobject thisObj, jbyteArray inJNIBytes);
```

Para la función principal del motor DPI:

```
long* netAnalizer(void *bytes, int numBytes, int* numProtocols);
```

4.3.2. Integración entre Java y motor DPI

La integración de ambas soluciones consistió en, con las modificaciones descritas en la sección anterior sobre el motor DPI, realizar una aplicación en Java que leyese un fichero de flujos de red y que ejecute todos los pasos del motor DPI, es decir, genere las DFA en base a las expresiones regulares y una vez generadas clasificase los flujos y devuelva los resultados a la aplicación en Java, que leyendo el fichero de configuración de protocolos de red debe ser capaz de establecer la relación entre lo devuelto por la función analizadora y el protocolo al que corresponde cada una de las repeticiones.

Una vez realizada la implementación, se obtuvo el siguiente error a la hora de ejecutar

la aplicación:

```
[oscar@compute-1-0 JNICuda]$ java -Xms1024m -Xmx81920m -Xss515m -Djava.library.path=. Test
Exception in thread "main" java.lang.UnsatisfiedLinkError: Readbinary.generaEntrada([B)[J
        at Readbinary.generaEntrada(Native Method)
        at Test.main(Test.java:11)
```

Figura 4.4: Error de linkado de la función

La JVM nos marca el error UnsatisfiedLinkError. Este error significa que no ha sido posible ejecutar la función ya que no la encuentra en la librería dinámica. Sin embargo, como se ha descrito en el apartado anterior, el prototipo de la función que trata de ejecutar la JVM es la misma que hemos implementado nosotros ¿cómo es posible que no la encuentre? La respuesta es el name mangling.

Name mangling es una característica del compilador de C++ para permitir la sobrecarga de métodos. La sobrecarga de métodos consiste en permitir declarar funciones con el mismo nombre pero con distintos parámetros. Sin embargo en la tabla de símbolos de la librería los nombres de las funciones deben de ser únicos para poder referenciarse correctamente, por lo que los nombres de las funciones que genera el compilador de C++ se modifican para ser 'únicos', de forma que dos funciones con el mismo nombre generarán dos símbolos distintos en la libreria. El name mangling de una función depende del nombre original de la función, de los parámetros que recibe y devuelve y del compilador que la genera.

```
000000000001fca0 T _Z29Java_Readbinary_generaEntradaP7JNIEnv_P8_jobjectP11_jbyteArray
000000000001f940 T _Z34__device_stub__Z13reduce_kernelmPhmPh
000000000001f010 T _Z37__device_stub__Z12match_kernelmPhS_PjmPhS_Pj
000000000001f240 T _Z46__device_stub__Z16match_kernel_phmmmPhPiPjS1_jmmPhPiPjS1_j
000000000001f500 T _Z48__device_stub__Z24transposeNoBankConflictsPhS_iiPhS_ii
000000000001f730 T _Z49__device_stub__Z25transposeNoBankConflicts2PhS_iiPhS_ii
0000000000016600 W _Z5fatalPc
000000000000b1a0 T _Z6tree_wP6wgraphj
00000000000097f0 T _Z7kruskalP6wgraphS0_P9partitionPiib
000000000001abe0 T _Z8load_dfaPKc
000000000000dff0 T _Z9check_depPjjj
```

Figura 4.5: Función con Name Mangling

Como podemos ver en 4.3.2, el nombre de la función en la tabla de simbolos de la librería es:

```
_Z29Java_Readbinary_generaEntradaP7JNIEnv_P8_jobjectP11_jbyteArray
```

Si todos los objetos se encuentran compilados con el compilador de C++ esto no supone un problema, sin embargo el programa que sirve de interfaz entre Java y el motor

DPI se encuentra codificado en C y, como el compilador de este lenguaje no soporta esa característica ya que C no permite sobrecargar funciones, al intentar ejecutar la función del motor DPI no la linka con su nombre modificado por el compilador de C++ y obtenemos el error que nos muestra la JVM. Esto se soluciona indicando delante de todos los prototipos de las funciones que sean llamadas desde un módulo escrito en C la directiva "extern 'C' ". Esta directiva indica al compilador de C++ que al símbolo que se va a generar para esa función no se le debe modificar su nombre, permitiendo su linkado desde un módulo codificado en C.

Con esta modificación, el nuevo prototipo para la función que ejecuta la clasificación de flujos quedaría:

```
extern 'C' long* netAnalizer(void *bytes, int numBytes, int*
    ↪ numProtocols);
```

```
0000000000020a80 T Java_Readbinary_generaEntrada
000000000022bbf0 B VERBOSE
                 U _IO_getc@@GLIBC_2.2.5
                 U _IO_putc@@GLIBC_2.2.5
                 w _Jv_RegisterClasses
                 U _Unwind_Resume@@GCC_3.0
0000000000001aff0 T _Z10create_dfaPKc
0000000000001fed0 T _Z10print_fluxP8_IO_FILER9flux_info
0000000000001f390 T _Z12match_kernelmPhS_Pj
0000000000001fc80 T _Z13reduce_kernelmPh
```

Figura 4.6: Función sin Name Mangling

4.3.3. Validación de la adaptación

Para validar esta implementación se tomó como referencia la aplicación original del clasificador de flujos. Puesto que únicamente se ha encapsulado la funcionalidad de clasificación de flujos en una librería dinámica para ejecutarse a través de JNI, al clasificar un mismo flujo de red con ambas aplicaciones los resultados deberían ser similares.

Se utilizó un flujo de red de 40 MB de datos obtenido de los ficheros de pruebas de [21] ya que el número de trazas de red debía ser representativo pero no era crítico para la prueba. Además, tuvo que codificarse una pequeña solución en Bash que adaptase las salidas de ambos programas ya que mientras el clasificador de flujos original mostraba el protocolo de cada paquete de red, la adaptación a través de JNI mostraba el número total de paquetes que pertenecían a cada protocolo, por lo que el script realizado modifica la

salida de la aplicación original para obtener un formato de salida similar al generado por la adaptación.

Una vez clasificado el mismo flujo con ambas soluciones y comparadas las salidas, se validó la adaptación al coincidir la clasificación de ambos programas.

4.4. Hadoop y motor DPI

La integración de ambas soluciones se ha realizado en base a los desarrollos previos descritos anteriormente. Por una parte, se ha desarrollado una aplicación en Hadoop que es capaz de procesar un fichero binario a través de la interfaz JNI, y por otra parte se ha desarrollado una aplicación basada en Java que a través de la interfaz JNI es capaz de ejecutar el clasificador de flujos e interpretar los resultados generados para asociar los flujos analizados con su correspondiente protocolo.

La fase Mapper de Hadoop ha sido adaptada para que el "Record" de datos recibido sea directamente transferido al clasificador de flujos a través de la interfaz JNI. Una vez clasificados por el motor DPI, se obtiene la relación entre el retorno del clasificador y el protocolo de red al que pertenece el número de coincidencias, utilizándose el nombre del protocolo como clave y el número de repeticiones como valor que se pasan al Reducer.

En la fase Reducer únicamente se suman los valores de cada clave, de forma que en la salida del MapReduce se mostrarán los protocolos encontrados en el flujo analizado y la cantidad de trazas que pertenecen a cada uno de esos protocolos.

Es necesario cargar en el HDFS todos los ficheros de configuración, DFAs y librerías dinámicas necesarias por el motor DPI, y configurar todos los paths de los ficheros de configuración para que en el caso de que necesiten de otro fichero lo busquen en el directorio en el que se encuentran y no en un subdirectorio, ya que la herramienta ToolRunner carga todos los ficheros en el mismo directorio del HDFS.

De esta forma, la arquitectura de la aplicación es la siguiente: se cargan en un directorio del HDFS todos los flujos de red que van a analizarse y se configura el tamaño de los "Records" de forma que sean múltiplo del tamaño total de bytes a analizar. El trabajo Hadoop se lanza indicando por parámetros los ficheros que van a ser cargados en el HDFS, que en nuestro caso son las librerías dinámicas y los ficheros de configuración y las DFAs de las expresiones regulares, junto con los path dentro del HDFS donde se encuentran los

ficheros de entrada y donde se van a escribir los ficheros de salida. Los ”Records” recibidos por el Mapper pasan directamente al clasificador de flujos, y su salida se utiliza como clave y valor para el Reducer que juntará todas las claves y sumará los valores para obtener el número de protocolos de cada tipo encontrado en las trazas del flujo de red.

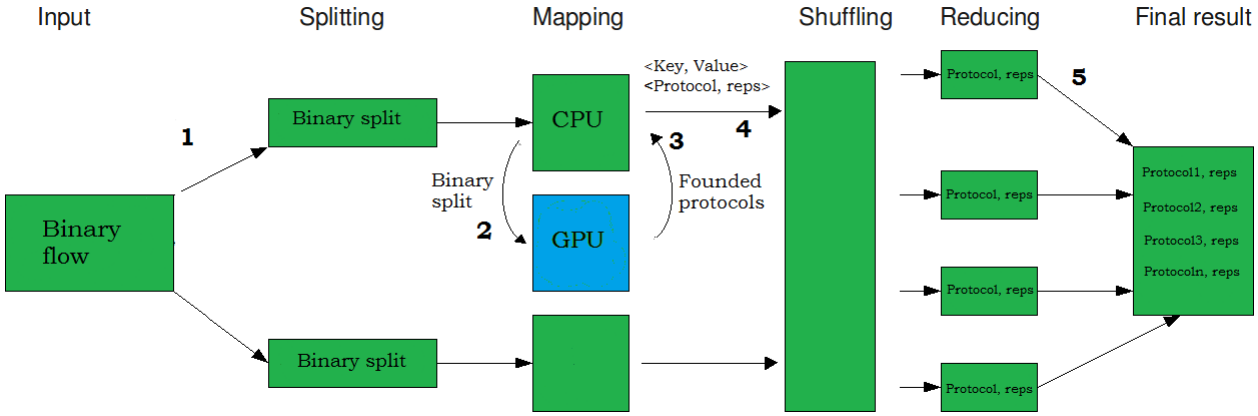

Figura 4.7: Arquitectura de la aplicación

5

Resultados

El método de desarrollo de la aplicación ha permitido ir realizando pruebas unitarias sobre las funcionalidades de forma que tenemos la certeza de que Hadoop es capaz de procesar ficheros binarios a través de la interfaz JNI, y el clasificador de flujos es capaz de clasificar via JNI los paquetes en función de las expresiones regulares con las que hemos generado las DFAs.

En esta sección se describe como se ha realizado la validación de la integración de ambas soluciones y se expone un test de rendimiento comparado la aplicación Hadoop + GPU con la aplicación original sobre GPU.

5.1. Validación de la implementación

La validación de la aplicación definitiva se ha realizado de forma similar a la realizada para la validación de la adaptación del clasificador de flujos a través de la intefaz JNI, es decir, clasificar un flujo a través de Hadoop y de la aplicación original y comprobar que ambas salidas coinciden. En esta ocasión se ha realizado una clasificación de un flujo de

300 MB de datos con el fin de realizar una verificación mas exahustiva, para ello:

- Se ha utilizado el analizador de redes Wireshark para capturar paquetes de la interfaz de red local en un entorno de alta densidad de tráfico de audio y video. Estas trazas se han guardado en un fichero de tipo pcap.

- El fichero pcap se trata con el programa FlowProcess[21]. Este programa genera en base a un fichero pcap un fichero de flujos que es analizable por el clasificador de flujos de red. Este fichero de flujos generado es el que se usará como entrada para la aplicación Hadoop.

- Se ha realizado la ejecución del clasificador de flujos original y de la aplicación Hadoop, y de la misma forma que se hizo al validar la implementación del clasificador a través de JNI, se parsea la salida del clasificador original para que el formato sea similar en ambas aplicaciones.

Al comparar ambas salidas obtenemos unos datos similares, lo que valida la solución Hadoop

5.2. Rendimiento

El objetivo de este proyecto es el de comprobar la viabilidad de combinar el framework Hadoop con aceleración de procesamiento sobre GPUs para clasificar paquetes de red, sin embargo puesto que el fin último de validar esta posibilidad es que se puedan diseñar aplicaciones que combinando ambas tecnologías se obtenga un mayor rendimiento que usando Hadoop o GPUs de manera única, se ha diseñado una prueba de rendimiento comparando el motor DPI original con su implementación junto a Hadoop.

5.2.1. Pruebas

Las pruebas que se han realizado han consistido en ejecutar ficheros iguales en las dos aplicaciones, aumentando gradualmente su tamaño para analizar el impacto que tiene la cantidad de paquetes a clasificar en el rendimiento de la aplicación.

Para medir el tiempo de ejecución, se ha ejecutado cada aplicación diez veces para mitigar los picos y valles de procesamiento de las máquinas que puedan desvirtuar las mediciones realizadas. Se ha utilizado la funcionalidad "time"de Linux, que nos muestra

el tiempo transcurrido desde la creación hasta la finalización del proceso, de forma que no medimos solo la clasificación de flujos si no toda la creación y destrucción del contexto del proceso clasificador.

La aplicación Hadoop se ha configurado de forma que se generen "Records" del tamaño de la mitad de los datos de entrada y de un tercio de los datos de entrada. Se ha elegido esta configuración en base a que la ejecución sobre GPU necesita un volúmen de datos suficiente como para que no penalice la transmisión entre la memoria de la GPU y la CPU, y a que el clasificador de flujos genera un overhead de unos diez segundos en cargar en memoria las DFAs de las expresiones regulares.

5.2.2. Resultados

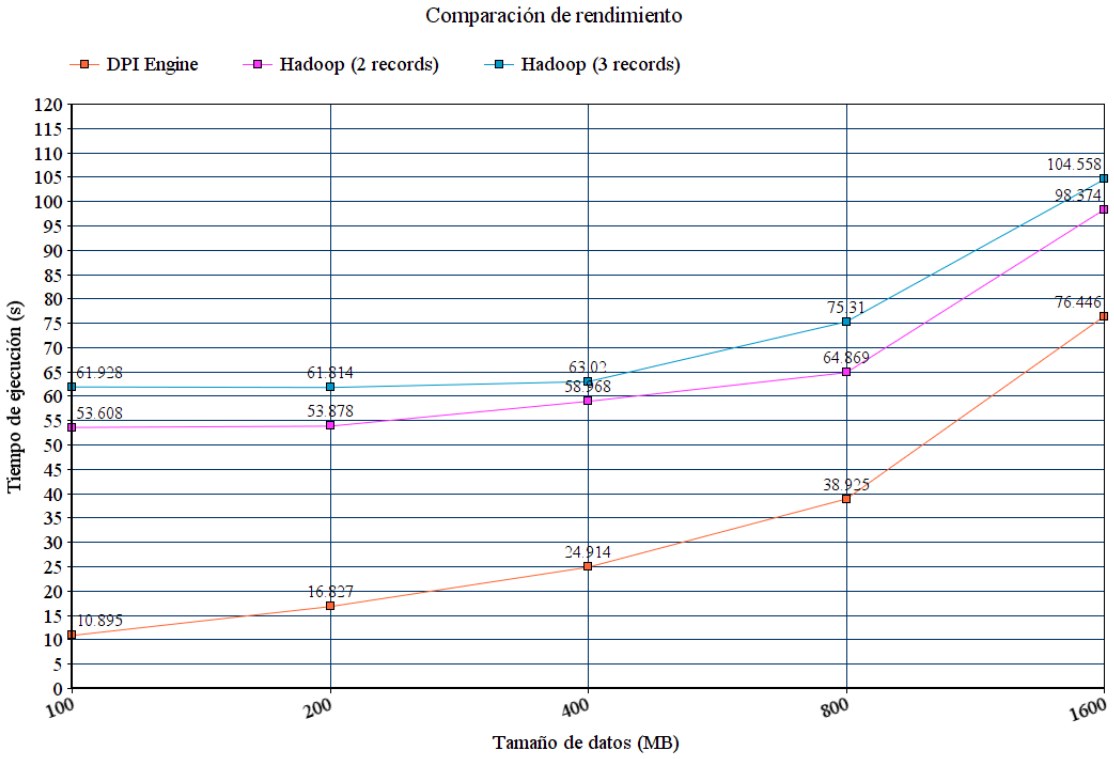

Figura 5.1: Pruebas de rendimiento

Antes de analizar los resultados, es necesario tener en cuenta varias consideraciones:

1) Para realizar las pruebas, se han creado previamente las DFAs de las expresiones

regulares de los protocolos a analizar. Esto es debido a que la fase de generación de las DFAs puede añadir una penalización de varios minutos en función de las expresiones que se utilicen para clasificar, y puede evitarse generándolas y pasándolas como parámetros a las aplicaciones de forma que la única penalización que se tiene es al cargarlas en memoria.

2) El clasificador de flujos desarrollado en C/C++ utiliza una gran cantidad de memoria de proceso tanto física como virtual. Esto afecta directamente a la aplicación Hadoop, puesto que las funcionalidades nativas se ejecutan dentro de la JVM y la memoria de la que dispone es limitada. Los nodos del clúster poseen 4 GB de memoria RAM, de la cual Hadoop tiene restringido el uso de 3 GB, lo que provoca que el mayor tamaño de ”Record” que pueden procesarse sea de 800 MB de datos, en caso de que sea de mayor tamaño provocan que se superen los 3 GB que permite reservar Hadoop de memoria para la JVM y aborte automáticamente la ejecución el trabajo.

Como se puede observar, la diferencia de rendimiento entre Hadoop y la aplicación nativa es del 750 % para ficheros de datos pequeños, pero conforme va aumentando el tamaño de los datos a procesar, el coste en tiempo de la aplicación nativa va aumentando a una mayor velocidad que el de la aplicación ejecutada en Hadoop, reduciéndose considerablemente su diferencia. Se puede observar en la gráfica que conforme aumenta la cantidad de datos a procesar los tiempos de ejecución tienden a estrecharse, por lo que si esta tendencia se mantiene llegará un tamaño de datos en el cuál sea más óptimo a nivel de tiempo de ejecución el utilizar la solución combinada que la aplicación nativa, justificando el desarrollo de la aplicación. Además se puede observar como el tiempo de ejecución con tres ”Records” de datos es mayor que con dos, debido a que la cantidad de datos a procesar no compensa todo el coste producido por la sincronización de un nodo de procesamiento más en el trabajo Hadoop.

Sin embargo, las pruebas no pueden considerarse concluyentes puesto que sería necesario haber utilizado entradas de varios GB de datos, únicamente pueden considerarse como una primera aproximación al rendimiento de ambas aplicaciones con un pequeño número de datos de entrada.

6

Conclusiones

Este proyecto ha supuesto una introducción al framework de Hadoop y a su combinación con el uso de GPUs para aumentar la capacidad de procesamiento y reducir el tiempo de ejecución de la aplicación.

Se ha visto como la arquitectura de Hadoop permite optimizar el procesamiento en clústers de servidores gracias a sus sistema de archivos distribuido y su gestión de los recursos hardware disponibles, y como toda esta gestión de recursos es transparente para el desarrollador de aplicaciones, que únicamente debe de centrarse en la codificación de la aplicación a través de la API de desarrollo. Estas cualidades hacen del framework Hadoop una buena solución para desarrollar aplicaciones orientadas al procesamiento de datos masivos que puedan ser procesados según la filosofía MapReduce.

Además, una aplicación Hadoop permite ejecutar funcionalidades nativas mediante el uso de librerías dinámicas cargadas en el sistema de archivos distribuido y la interfaz JNI. Estas funcionalidades nativas nos permiten, entre otras muchas cosas, la programación de aplicaciones que se ejecuten sobre la GPU a través de las APIs y los compiladores propios de cada una de las distintas GPUs del mercado.

Hemos comprobado la viabilidad de combinar ambas tecnologías para diseñar un clasificador de paquetes de red, sin embargo las posibilidades de estas dos tecnologías son muy diversas ya que cualquier aplicación que pueda desarrollarse siguiendo el paradigma MapReduce es susceptible de utilizar GPUs para acelerar el procesamiento de datos.

Antes de diseñar una aplicación que combine ambas soluciones es necesario tener presente la cantidad de datos que se van a procesar. Hadoop es un framework que realiza splits de los datos de entrada, y estos splits de datos deben ser lo suficientemente grandes como para que el overhead generado por la transferencia de los datos de la CPU a la GPU en ambos sentidos[19] y el overhead generado por el propio framework de Hadoop para configurar todos los parámetros y nodos del sistema distribuido sea compensado por la mejora obtenida por la mayor velocidad de procesamiento de la GPU frente a la CPU y el que varios nodos procesen datos en paralelo, lo que implica que el tamaño de los datos debe ser como mínimo del orden de Gigabytes.

6.1. Trabajo futuro

Se proponen tres alternativas para desarrollar en el futuro partiendo de este proyecto:

1- El principal cuello de botella de esta implementación es el hecho de tener que cargar las DFAs cada vez que se ejecuta el clasificador de flujos, penalizando varios segundos cada llamada al clasificador. Realizar una adaptación del motor DPI para que únicamente se cargen una única vez las DFA en la primera llamada al clasificador de flujos, de forma que se mantengan estáticamente en memoria para posteriores llamadas.

2- Adaptar el clasificador de flujos para funcionar a través de otro framework distinto a JNI, como podría ser JCUDA, de forma que se puedan comparar rendimientos con el objetivo de comprobar cual es el framework más interesante para combinar Hadoop y GPU en términos de rendimiento.

3- Desarrollar otras aplicaciones basadas en esta implementación para valorar el impacto de combinar ambas soluciones a nivel de rendimiento.

Bibliografía

[1] *A Performance Study of General Purpose Applications on Graphics Processors*. S. Che, J. Meng, J. W. Sheaffer, K. Skadron, 2008. URL: http://www.cs.virginia.edu/~skadron/Papers/GPGPU_workshop_CUDA.pdf.

[2] *MapReduce: Simplified Data Processing on Large Clusters*. J. Dean, S. Ghemawat, 2004. URL: http://static.googleusercontent.com/media/research.google.com/es//archive/mapreduce-osdi04.pdf.

[3] *Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units*. Q. Fang y D. A. Boas, 2009. URL: https://www.osapublishing.org/oe/fulltext.cfm?uri=oe-17-22-20178&id=187243.

[4] *An Implementation of GPU Accelerated MapReduce: Using Hadoop with OpenCL for Data- and Compute-Intensive Jobs*.

[5] M. Razip K. Xu A. Sabne A. Mujahid. *MapReduce on GPUs*. 2012. URL: https://engineering.purdue.edu/~eigenman/ECE563/ProjectPresentations/GPUAcceleratedHadoop.pdf (visitado 02-05-2015).

[6] *Mars: A MapReduce Framework on Graphics Processors*. B. He, W. Fang, N. K. Govindaraju, Q. Luo, T. Wang, 2008. URL: http://www.cse.ust.hk/gpuqp/Mars_tr.pdf.

[7] *Evaluating MapReduce for Multi-core and Multiprocessor Systems*. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, C. Kozyrakis, 2007. URL: http://csl.stanford.edu/~christos/publications/2007.cmp_mapreduce.hpca.pdf.

[8] *MapCG: Writing Parallel Program Portable between CPU and GPU*. C. Hong, D. Chen, W. Chen, W. Zheng, H. Lin, 2010. URL: http://www.cse.ust.hk/gpuqp/Mars_tr.pdf.

[9] OpenMP Architecture Review Board. *OpenMP*. 1997. URL: http://openmp.org/wp/ (visitado 05-05-2015).

[10] *Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System*. R. M. Yoo, A. Romano, C. Kozyrakis, 2009. URL: http://csl.stanford.edu/~christos/publications/2009.scalable_phoenix.iiswc.pdf.

[11] Amazon. *Amazon Elastic MapReduce*. 2009. URL: http://aws.amazon.com/es/elasticmapreduce/ (visitado 02-05-2015).

[12] Amazon. *Amazon Elastic MapReduce Announces Support for Cluster Compute and Cluster GPU Instances*. 2010. URL: http://aws.amazon.com/es/about-aws/whats-new/2010/11/15/amazon-elastic-mapreduce-announces-support-for-cluster-compute-and-cluster-gpu-instances/ (visitado 02-05-2015).

[13] Wikipedia. *Hadoop*. 2015. URL: https://es.wikipedia.org/wiki/Hadoop (visitado 05-05-2015).

[14] JCuda. *JCuda*. 2009. URL: http://www.jcuda.org/ (visitado 05-05-2015).

[15] Open Source AMD. *Java Aparapi*. 2011. URL: http://developer.amd.com/tools-and-sdks/opencl-zone/aparapi/ (visitado 05-05-2015).

[16] Oracle. *Java Native Interface*. 1999. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/ (visitado 05-05-2015).

[17] Apache. *Hadoop*. 2015. URL: http://hadoop.apache.org/ (visitado 05-05-2015).

[18] *A Comparative Analysis of Microarchitecture Effects on CPU and GPU Memory System Behavior*. J. Hestness, S. W. Keckler, D. A. Wood, 2014. URL: http://research.cs.wisc.edu/multifacet/papers/iiswc14_characterize.pdf.

[19] *Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer*. C. Gregg, K. Hazelwood, 2011. URL: http://www.cs.virginia.edu/kim/docs/ispass11.pdf.

[20] M. Strait J. Levandoski E. Sommer. *L7-filter*. 2009. URL: http://l7-filter.sourceforge.net/ (visitado 10-05-2015).

[21] *Clasificacion de flujos en 10 Gbps ethernet mediante Intel DPDK y GPUs*. R. Leira, 2013. URL: http://ir.ii.uam.es/~fdiez/TFGs/gestion/leidos/2013/20130611RafaelLeiraOsuna.pdf.

[22] Proyectos Ágiles. *Desarrollo iterativo e incremental*. 2015. URL: http://www.proyectosagiles.org/desarrollo-iterativo-incremental (visitado 05-05-2015).

[23] Apache Hadoop. *Hadoop Pipes*. 2015. URL: https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/mapred/pipes/package-summary.html (visitado 08-05-2015).

[24] Apache Hadoop. *Hadoop Streaming*. 2015. URL: http://hadoop.apache.org/docs/current/hadoop-streaming/HadoopStreaming.html (visitado 08-05-2015).

[25] D. Thiebaut. *Running C++ Programs on Hadoop*. 2010. URL: http://cs.smith.edu/dftwiki/index.php/Hadoop_Tutorial_2.2_--_Running_C++_Programs_on_Hadoop/ (visitado 12-05-2015).

Apéndices

A

Tabla de rendimiento

Aplicación	**100 MB**	**200 MB**	**400 MB**	**800 MB**	**1600 MB**
Motor DPI	0m10.895s	0m16.827s	0m24.914s	0m38.925s	1m16.446s
Hadoop 2 splits	0m53.608s	0m53.878s	0m58.968s	1m4.869s	1m38.374s
Hadoop 3 splits	1m1.928s	1m1.814s	1m3.020s	1m15.318s	1m44.558s

Tabla A.1: Rendimiento Hadoop vs aplicación nativa